

## Algorithm Analysis

For our AVL Tree Algorithm Analysis, we will consider any helper functions called as well as consider the process of comparing a string with no limit on characters. Ideally, we are on the order of logarithmic time for searching, inserting, and deleting nodes from the tree.

### insert *NAME ID*

- Public Member Function: *void insert(const string name, const string iD);*
- Private Helper Function(s): *GatorNode\* insert(GatorNode\* localRoot, const string name, const string iD);* **\*\*All the rotation functions\*\***
- Variables: Length of NAME string  $\equiv k$ , Number of nodes  $\equiv n$

The length of the NAME string does not affect this function as it traverses the binary tree using ID as a comparable. ID is fixed in length. The number of nodes to compare to is cut in half each time we make a comparison.

$$\mathcal{O}(\log n)$$

### remove *ID*

- Public Member Function: *void remove(const string iD);*
- Private Helper Function(s): *void remove(GatorNode\* localRoot, const string iD); GatorNode\* inOrderSuc(GatorNode\* localRoot);*
- Variables: Length of NAME string  $\equiv k$ , Number of nodes  $\equiv n$

Again, because we are using ID as a fixed length string for comparison, we are able to stay on logarithmic time complexity.

$$\mathcal{O}(\log n)$$

### search *ID*

- Public Member Function: *void search(const string findMe);*
- Private Helper Function(s): *GatorNode\* search(GatorNode\* localRoot, const string findMe, bool found);*
- Variables: Length of NAME string  $\equiv k$ , Number of nodes  $\equiv n$

Searching based on ID is dependent only on the number of nodes as the dataset grows. Still on  $\log(n)$  time.

$$\mathcal{O}(\log n)$$

## search *NAME*

- Public Member Function: *void search(const string findMe);*
- Private Helper Function(s): *GatorNode\* search(GatorNode\* localRoot, const string findMe, bool found);*
- Variables: Length of NAME string  $\equiv k$ , Number of nodes  $\equiv n$

Searching based on NAME incorporates the number of characters of the NAME string as a factor in scaling. As we traverse the tree we have to make comparisons between NAMES, and because we are not sorted by NAME we have to check every node and compare against  $k$  characters.

$$\mathcal{O}(k * n)$$

## printInorder

- Public Member Function: *void printInOrder();*
- Private Helper Function(s): *void printInOrder(GatorNode\* localRoot, bool flag) const;*
- Variables: Length of NAME string  $\equiv k$ , Number of nodes  $\equiv n$

All of our printing functions for the tree will be linear in time complexity. We have to recursively visit every node to print it.

$$\mathcal{O}(n)$$

## printPreorder

- Public Member Function: *void printPreOrder();*
- Private Helper Function(s): *void printPreOrder(GatorNode\* localRoot, bool flag) const;*
- Variables: Length of NAME string  $\equiv k$ , Number of nodes  $\equiv n$

$$\mathcal{O}(n)$$

## printPostorder

- Public Member Function: *void printPostOrder();*
- Private Helper Function(s): *void printPostOrder(GatorNode\* localRoot, bool flag) const;*
- Variables: Length of NAME string  $\equiv k$ , Number of nodes  $\equiv n$

$$\mathcal{O}(n)$$

## printLevelCount

- Public Member Function: *void printLevelCount() const;*
- Private Helper Function(s):
- Variables: Length of NAME string  $\equiv k$ , Number of nodes  $\equiv n$

Note that the implementation of this function simply requires looking at the height of the root member variable, however balancing at every node is required to keep this number up to date. Hence, it is the same affect as visiting every node, so linear time similar to traversal.

$$\mathcal{O}(n)$$

## removeInOrder *N*

- Public Member Function: *void removeInOrder(unsigned int n); // uses inorder traversal*
- Private Helper Function(s): *void inOrder(GatorNode\* localRoot, unsigned int n, string toReturn); \*\*Also calls private remove function\*\**
- Variables: Length of NAME string  $\equiv k$ , Number of nodes  $\equiv n$

This member function requires every node to be visited until the Nth node is found. Hence it is big O linear time.

$$\mathcal{O}(n)$$

## Summary

This was one of the more challenging assignments I have done for my Computer Science degree. I learned a lot and gained a much deeper appreciation for how much work programmers before me have put in to make efficient data structures for the rest of us to use. Specifically, I learned a ton about implementation through this project. Testing of edge cases, toying around with different function/return types, and keeping tidy memory for dynamically allocated objects were all great subjects to gain knowledge on.

I am very happy with the way the project turned out, however I would change the way that I handle printing, traversals, and searching in this project if I were to start over. I realized after I had already done a lot of the heavy lifting in my project that many of the `std::cout` calls and much of the logic on traversing were buried layers down in my AVL Tree class. This created problems as I added on future functionality because my code was not super modular. If I had separated out my traversal functions and set my search and print functions to instead use references / pointers as return types I would not have had so many issues with refactoring code and setting up unit tests. Overall, though, very cool project!